

# PHarr XCPC Templates

PHarr

2023 年 4 月 27 日



# 目录



# 第一章 编程技巧和基础算法

## 1.1 读入优化

```
1 // 快读
2 int read() {
3     int x = 0, f = 1, ch = getchar();
4     while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
5     if (ch == '-') f = -1, ch = getchar();
6     while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
        ch = getchar();
7     return x * f;
8 }
9 // 关闭同步
10 ios::sync_with_stdio(false);
11 cin.tie(nullptr);
```

## 1.2 C++ 标准输入输出

```
1 // 设置输出宽度为 x
2 cout << setw(x) << val;
3
4 // 设置保留小数位数 x , 并四舍五入
5 cout << fixed << setprecision(x) << val;
6
7 //按进制输出
8 cout << bitset<10>(i); // 二进制
9 cout << oct << i; // 八进制
10 cout << dec << i; // 十进制
11 cout << hex << i; // 十六进制
12
```

```

13 // 设置填充符
14 cout << setw(10) << 1234; // 默认是空格
15 cout << setw(10) << setfill('0') << 1234; // 设置填充符
16 //左侧填充
17 cout<<setw(10)<<setfill('0')<<setiosflags(ios::left)<<123;
18 //右侧填充
19 cout<<setw(10)<<setfill('0')<<setiosflags(ios::right)<<123;
20
21 // 单个字符
22 ch = cin.get();
23 cout.put(ch);
24
25 // 指定长度字符串读入
26 cout.get(str,80,'a');// 字符串 字符个数 终止字符
27
28 // 整行读入
29 getlin( cin , s );

```

## 1.3 倍增

天才 ACM

给定一个整数  $M$ ，对于任意一个整数集合  $S$ ，定义“校验值”如下：

从集合  $S$  中取出  $M$  对数 (即  $2 \times M$  个数，不能重复使用集合中的数，如果  $S$  中的整数不够  $M$  对，则取到不能取为止)，使得“每对数的差的平方”之和最大，这个最大值就称为集合  $S$  的“校验值”。

现在给定一个长度为  $N$  的数列  $A$  以及一个整数  $T$ 。我们要把  $A$  分成若干段，使得每一段的“校验值”都不超过  $T$ 。求最少需要分成几段。

1. 初始化  $p = 1$  ,  $r = 1 = 1$
2. 求出  $[1, r+p]$  这一段的校验值，若校验值小于等于  $T$  则  $r += p, p *= 2$ ，否则  $p /= 2$
3. 重复上一步知道  $p$  的值变为 0 此时的  $r$  即为所求

```

1 #include<bits/stdc++.h>
2 #define int long long
3 using namespace std;
4
5 int read() {
6     int x = 0, f = 1, ch = getchar();

```

```

7   while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
8   if (ch == '-') f = -1, ch = getchar();
9   while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
      ch = getchar();
10  return x * f;
11 }
12
13 const int N = 5e5+5;
14 int a[N] , b[N];
15 int n , m , t , res;
16 int query( int l , int r ){
17     if( r > n ) return 1e19;
18     for( int i = l ; i <= r ; i ++ ) b[i] = a[i];
19     sort( b + l , b + r + 1);
20     int ans = 0;
21     for( int i = l , j = r , t = 1 ; t <= m && i < j ; t ++ , i ++ , j
        -- )
22         ans += ( b[i] - b[j] ) * ( b[i] - b[j] );
23     return ans;
24 }
25
26 void solve(){
27     n = read() , m = read() , t = read() , res = 0;
28     for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
29     for( int l = 1 , r = 1 , p; l <= n ; l = r + 1 ){
30         p = 1 , r = l;
31         while( p ){
32             if( query( l , r + p ) <= t ) r += p , p *= 2;
33             else p /= 2;
34         }
35         res ++;
36     }
37     cout << res << "\n";
38 }
39
40 int32_t main() {
41     for( int T = read(); T ; T -- ) solve();
42     return 0;

```

43 }

## 1.4 语法杂项

### 1.4.1 define 与 typedef

typedef是用来给类型定义别名，是编译器处理的

#define是字面上进行宏定义用的，是在预处理阶段使用的

```
1 #define STRING char * //宏定义
2 STRING name , sign;//声明
3 char * name , sign;//会被替换成这种的结果,只有 name 是指针
4 // 所以定义类型时候应该回避 #define 而采用typedef
5 typedef char * STRING;
6 char * name , * sign;//会被替换成这种结果
```

### 1.4.2 fill 数组填充

```
1 // 一维数组
2 int a[5];
3 fill( a , a + 5 , 3 );
4
5 // 二维数组
6 int a[5][4];
7 fill( a[0] , a[0] + 5 * 4 , 6 );
8
9 // vector
10 vector<int> a;
11 fill( a.begin() , a.end() , 9 );
```

## 1.5 Lambda 表达式

以下内容绝大部分使用与c++14及更新的标准

Lambda 的组成部分是

```
1 [capture] (parameters) mutable -> return-type {statement};
```

首先capture是捕获列表可以从所在代码块中捕获变量。

什么都不写[]就是不进行任何捕获，[=]是值捕获，[&]是引用捕获，值捕获不能修改变量的值，引用捕获可以。特别的，如果值捕获希望在函数内部修改可以使用mutable关键字



同时捕获列表也可以单独针对某一个变量[a]、[&a]分别是值捕获和引用捕获。当然也可以混用[=,&a]对所有变量值捕获，但a除外，a是引用捕获。

然后就是parameters参数列表和statement函数主体，这里与普通的函数没有区别。

-> return-type, 函数范围值类型，如果不写可以自动推断，但是如果有多多个return且返回类型不同就会CE

c++14之后可以用auto来自动的把函数赋值给变量，c++11中则需要自己写

## 1.6 MT 19937

mt19937是一个很便捷的随机数生成算法，在 c++11中使用非常便捷

```
1 mt19937 rd(seed) ; // 这样就填入了一个随机数种子
2 rd(); // 这样就会返回一个随机数
3 mt19937_64 rd(); // 相同用法，不过返回是一个 64 位整形
```

## 1.7 常用常数

在<math.h>库中有一些常用的参数

```
1 #if defined _USE_MATH_DEFINES && !defined _MATH_DEFINES_DEFINED
2     #define _MATH_DEFINES_DEFINED
3     #define M_E          2.71828182845904523536    // e
4     #define M_LOG2E      1.44269504088896340736    // log2(e)
5     #define M_LOG10E     0.434294481903251827651    // log10(e)
6     #define M_LN2        0.693147180559945309417    // ln(2)
7     #define M_LN10       2.30258509299404568402    // ln(10)
8     #define M_PI         3.14159265358979323846    // pi
9     #define M_PI_2       1.57079632679489661923    // pi/2
10    #define M_PI_4        0.785398163397448309616    // pi/4
11    #define M_1_PI        0.318309886183790671538    // 1/pi
12    #define M_2_PI        0.636619772367581343076    // 2/pi
13    #define M_2_SQRTPI    1.12837916709551257390    // 2/sqrt(pi)
14    #define M_SQRT2       1.41421356237309504880    // sqrt(2)
15    #define M_SQRT1_2     0.707106781186547524401    // 1/sqrt(2)
16 #endif
```

但是<math.h>并没有默认定义\_USE\_MATH\_DEFINES，所以用之前需要先定义（在万能头下貌似已经被定义过了），在开头加上#define \_USE\_MATH\_DEFINES即可

## 1.8 有用的库函数

1. `fabs(x)` 用于计算浮点数的绝对值
2. `exp(x)` 计算  $e^x$
3. `log(x)` 计算  $\ln(x)$
4. `__lg(x)` 计算  $\lfloor \log_2(x) \rfloor$
5. `__gcd(x,y)` 计算  $\gcd(x,y)$ , 不过在 C++17 引入了 `gcd(x,y)`, `lcm(x,y)`
6. `ceil(x)` 返回  $\lceil x \rceil$
7. `floor(x)`  $\lfloor x \rfloor$
8. `next_permutation(begin,end)` 将  $[begin, end)$  变为下一个排列, 如果已经是最后一个排列就返回 0
9. `prev_permutation(begin,end)` 将  $[begin, end)$  变为上一个排列, 如果已经是最后一个排列就返回 0
10. `shuffle(begin,end,gen)` 打乱  $[begin, end)$ , `gen` 是一个随机数生成器 (参考 mt19937)
11. `is_sorted(begin,end)` 判断是否升序排序
12. `max(l), min(l)` 对于数组或列表返回最大最小值, 例 `max({x,y,z})`
13. `exp2(x)` 计算  $2^x$
14. `log2(x)` 计算  $\log_2(x)$
15. `hypot(x,y)` 计算  $\sqrt{x^2 + y^2}$

## 1.9 交互题

这里说的交互题只是 STDIO 交互题

交互题往往是不会限制运行时间的, 一般是通过限制与 oj 的交换次数。对于 IO 交换的题目, 要注意的是在每一次输出后都必须刷新输出缓冲后才可以读入。下面介绍几种语言如何刷新缓冲。

1. C `fflush(stdout)`
2. C++ `fflush(stdout)` 或者 `cout << flush` 或者用 `cout << endl` 输出换行也会自动刷新
3. Java `System.out.flush()`
4. Python `stdout.flush()`

## 1.10 Windows 下的对拍器

```
1 :again
2 data.exe > data.in
3 std.exe < data.in > std.out
4 test.exe < data.in > test.out
5 fc std.out test.out
6 if not errorlevel 1 goto again
```



## 第二章 数据结构

### 2.1 并查集

```
1 // 初始化
2 for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
3 // 查找
4 int getFa( int x ){
5     if( fa[x] == x ) return x;
6     return fa[x] = getFa( fa[x] );
7 }
8 // 合并
9 void merge( int x , int y ){
10     fa[getFa(x) ] = getFa(y);
11 }
12
13 /*
14  * 下面是一种按秩合并的写法
15  * 简单来说fa[x]<0表示该点为根结点，当x为根节点时 fa[x] = -size[x]
16  */
17 class dsu{
18 private:
19     vector<int> fa;
20 public:
21     dsu( int n = 1 ){
22         fa = vector<int>( n+1 , -1 ) , fa[0] = 0;
23     }
24     int getfa( int x ){
25         if( fa[x] < 0 ) return x;
26         return fa[x] = getfa( fa[x] );
27     }
28     void merge( int x , int y ){
```

```

29         x = getfa(x) , y = getfa(y);
30         if( x == y ) return ;
31         if( fa[x] > fa[y] ) swap( x , y );
32         fa[x] += fa[y] , fa[y] = x;
33     }
34     bool check( int x , int y ){
35         x = getfa(x) , y = getfa(y);
36         return ( x == y );
37     }
38 };

```

## 2.2 链式前向星

链式前向星又名邻接表，其实现在我已经几乎不会再手写链式前向星而是采用vector来代替

```

1 vector<int> e[N]; // 无边权
2 vector< pair<int,int> > e[N]; 有边权
3
4 e[u].push_back(v); // 加边(u,v)
5 e[u].push_back( { v, w } ); //加有权边 (u,v,w)
6 // 无向边 反过来再做一次就好
7
8 for( auto v : e[u] ){ // 遍历
9 }
10 for( auto [ v , w ] : e[u] ) { // 遍历有权边
11 }

```

## 2.3 Hash

### 2.3.1 Hash 表

对数字的 hash

```

1 for( int i = 1 ; i <= n ; i ++ ) b[i] = a[i]; // 复制数组
2 sort( b + 1 , b + 1 + n ) , m = unique( b + 1 , b + 1 + n ) - b; // 排序去重
3 for( int i = 1 ; i <= n ; i ++ ) //hash
4     a[i] = lower_bound( b + 1 , b + 1 + m , a[i] ) - b;

```

除此之外，如果更加复杂的 hash 全部使用unordered\_map容器

### 2.3.2 字符串 Hash

把字符串当做一个  $p$  进制数，如果哈希冲突就双哈希

如果  $s = \text{"abc"}$  那么  $H(s) = a \times p^2 + b \times p + c$

```

1 //AcWing 138. 兔子与兔子
2 #include<bits/stdc++.h>
3 #define ull unsigned long long
4 using namespace std;
5
6 const int N = 1e6+5;
7 int n ;
8 ull Hash[N] , p[N] , P = 131;
9 string s;
10
11 int read() {...}
12
13 ull getHash( int l , int r ){
14     return Hash[r] - Hash[l-1] * p[ r - l + 1 ];
15 }
16
17 int32_t main() {
18     cin >> s;
19     n = s.size() , p[0] = 1;
20     for( int i = 1 ; i <= n ; i ++ )
21         Hash[i] = Hash[i-1] * P + (s[i-1]-'a'+1) , p[i] = p[i-1] * P;
22     for( int m = read() , l1 , r1 , l2 , r2 ; m ; m -- ){
23         l1 = read() , r1 = read() , l2 = read() , r2 = read();
24         if( getHash(l1,r1) == getHash(l2,r2) ) printf("Yes\n");
25         else printf("No\n");
26     }
27 }

```

## 2.4 栈

### 2.4.1 包含 Min 函数的栈

一个支持  $O(1)$  的  $push()$ ,  $pop()$ ,  $top()$ ,  $getmin()$  的栈

在维护栈的同时维护一个栈来保存历史上每个时刻都最小值

```

1 struct MinStack{
2     stack<int> a , b;
3     void push( int x ){
4         a.push(x);
5         if( b.size() ) b.push( min( x , b.top() ) );
6         else b.push(x);
7     }
8     void pop(){
9         a.pop() , b.pop();
10    }
11    int top(){
12        return a.top();
13    }
14    int getMin(){
15        return b.top();
16    }
17 };

```

### 2.4.2 单调栈

用来  $O(n)$  的维护出每一个点左侧第一个比他高的点

```

1 int h[N] , l[N]; // h[i] 用来记录每一个点的高度 l[i] 记录每一个左侧第
    一个比 i 高的点的位置
2 stack<int> stk;
3
4 h[0] = INF; //为了便于处理把 0 处理为正无穷
5 stk.push( h[0] );
6 for( int i = 1 ; i <= n ; i ++ ){
7     while( h[i] >= h[ stk.top() ] ) stk.pop();
8     l[i] = stk.top() , stk.push(i);
9 }

```

## 2.5 ST 表

ST 表解决的问题是没有修改且查询次数较多 ( $10^6$ ) 的区间最值查询

$f[i][j]$  表示  $i$  向后  $2^j$  个数的最大值

所以  $f[i][j] = \max( f[i][j-1] , f[ i + ( 1 \ll j-1 ) ][j-1]$

询问首先计算出数最大的  $x$  满足  $2^x \leq r - l + 1$



这样的话  $[l, r] = [l, l + 2^x - 1] \cup [r - 2^x + 1, r]$

```

1 // LOJ10119
2 const int N = 1e6+5 , logN = 20;
3 int a[N] , log_2[N] , f[N][ logN + 5 ];
4 int32_t main() {
5     int n = read() , m = read();
6     for( int i = 1 ; i <= n ; i ++ )
7         a[i] = read();
8     log_2[0] = -1; // 这样初始化可以使得 log_2[1] = 0
9
10    for( int i = 1 ; i <= n ; i ++ ) // O(n) 预处理边界条件 和 log2(i)
11        f[i][0] = a[i] , log_2[i] = log_2[i>>1] + 1;
12
13    for( int j = 1 ; j <= logN ; j ++ )
14        for( int i = 1; i + ( 1 << j ) - 1 <= n ; i ++ )
15            f[i][j] = max( f[i][j-1] , f[ i + ( 1 << j - 1 ) ][j-1] );
16
17    for( int l , r , s ; m ; m -- ){
18        l = read() , r = read() , s = log_2[ r - l + 1 ];
19        printf("%d\n" , max( f[l][s] , f[ r - ( 1 << s ) + 1 ][s] ));
20    }
21    return 0;
22 }
```

## 2.6 树状数组

### 2.6.1 单点修改，区间查询

```

1 // lowbit 函数
2 #define lowbit(x) ( x & -x )
3 // O(N) 递推建树
4 void build(){
5     for( int i = 1 , fa = i + lowbit(i) ; i <= n ; i ++ , fa = i +
6         lowbit(i) )
7         if( fa <= n ) bit[fa] += bit[i];
8 }
9 // 单点修改
10 void add( int x , int val ){
```

```

10     for( int i = x ; i <= n ; i += lowbit(i) )
11         bit[i] += val;
12 }
13 // 区间查询[1,x]的和
14 int getVal( int x ){
15     int res = 0;
16     for( int i = x ; i ; i -= lowbit(i) )
17         res += bit[i];
18     return res;
19 }

```

### 2.6.2 区间修改，单点查询

这里用线段树维护一下差分数组就好

```

1 // op == 1 [l,r] 加上 val
2 // op == 2 查询位置 l 的值
3 int32_t main() {
4     n = read() , m = read();
5     for( int i = 1 , x = 0 , lst = 0; i <= n ; i ++ ) x = read() , bit
        [i] = x - lst , lst = x ;
6     build();
7     for( int op , l , r , val; m ; m -- ){
8         op = read();
9         if( op == 1 ) l = read() , r = read() , val = read() , add( l
            , val ) , add( r + 1 , - val );
10        else l = read() , printf("%d\n" , getVal(l) );
11    }
12    return 0;
13 }

```

## 2.7 分块

```

1 // https://loj.ac/p/6280
2 #include <bits/stdc++.h>
3 using namespace std;
4 #define int long long
5
6 int read() {...}

```

```

7  class decompose {
8  private:
9      struct block {
10         int l, r, sum, tag; // tag是单点修改时的懒惰标记
11         vector<int> val;
12
13         block(int l, int r) : l(l), r(r) {
14             sum = tag = 0;
15             val = vector<int>();
16         }
17     };
18     int len;
19     vector<block> part;
20     vector<int> pos;
21 public:
22     decompose(vector<int> &v) {
23         len = v.size();
24         int t = sqrt(len);
25         pos = vector<int>(len + 1);
26         for (int i = 1; i <= t; i++) // 预处理区间信息
27             part.emplace_back((i - 1) * t + 1, i * t);
28         if (part.back().r < len) // 处理结尾零散部分
29             part.emplace_back(part.back().r + 1, len);
30         for (int i = 1, j = 0; i <= len; i++) {
31             if (i > part[j].r) j++;
32             part[j].val.emplace_back(v[i - 1]);
33             part[j].sum += v[i - 1], pos[i] = j;
34         }
35     }
36
37     int getSum(int l, int r) {
38         int sum = 0;
39         for (int i = pos[l]; i <= pos[r]; i++) {
40             if (part[i].l >= l && part[i].r <= r) sum += part[i].sum;
41             else
42                 for (auto j = max(l, part[i].l) - part[i].l; j <= min(
43                     r, part[i].r) - part[i].l; j++)
44                     sum += part[i].val[j] + part[i].tag;
45         }
46     }
47 };

```

```
44     }
45     return sum;
46 }
47
48 void update(int l, int r, int d) {
49     for (int i = pos[l]; i <= pos[r]; i++) {
50         if (part[i].l >= l && part[i].r <= r){
51             part[i].tag += d;
52             part[i].sum += part[i].val.size() * d;
53         }
54         else
55             for (int j = max(l, part[i].l) - part[i].l; j <= min(r
                    , part[i].r) - part[i].l; j++)
56                 part[i].val[j] += d, part[i].sum += d;
57     }
58 }
59 };
60
61
62 int32_t main() {
63     int n = read();
64     vector<int> a(n);
65     for (auto &i: a) i = read();
66     decompose p(a);
67     for (int opt, l, r, c, sum; n; n--) {
68         opt = read(), l = read(), r = read(), c = read();
69         if (opt == 0)
70             p.update(l, r, c);
71         else {
72             c++, sum = p.getSum(l, r), sum = (sum % c + c) % c;
73             printf("%lld\n", sum);
74         }
75     }
76     return 0;
77 }
```

## 2.8 差分

### 离散化差分

```
1 // 离散化差分这里我用map实现
2 map<int,int> b;
3
4 void update( int l , int r , int v ){
5     b[l] += v , b[r+1] -= v;
6 }
7
8 // 求和
9 for( auto it = b.begin() ; next(it) != b.end() ; it = next(it) )
10     next(it)->second += it->second;
```

### 2.8.1 二维前缀和、差分

```
1 // 二维前缀和
2 b[i][j] = b[i-1][j]+b[i][j-1]-b[i-1][j-1] + a[i][j];
3
4 // 求 (x1,y1) ~ (x2,y2) 和
5 b[x2][y2] - b[x2][y1-1] - b[x1-1][y2] + b[x1-1][y1-1];
```



## 第三章 图论

### 3.1 拓扑排序

在一个 DAG 中，将图中的顶点以线性的方式排序，使得对于任何一条  $u$  到  $v$  的有向边， $u$  都可以出现在  $v$  的前面

**拓扑序判环**如果图中已经没有入度为零的点但是依旧有点时，即说明图中存在环。

**拓扑序判链**如果求拓扑序的过程中队列中同时存在两个及以上的元素，说明拓扑序不唯一，不是一条链

**字典序最大、最小的拓扑序**把 Kahn 中队列换成是大根堆、小根堆实现的优先队列就好

#### 3.1.1 DFS 算法

```
1  vector<int> e[N]; // 邻接表
2  vector<int> topo; // 存储拓扑序
3  set<int> notInDeg; // 储存没有入读的点
4  int vis[N];
5
6  bool dfs( int u ){
7      vis[u] = 1;
8      for( auto v : e[u] ){
9          if( vis[v] ) return 0;
10         if( !dfs(v) ) return 0;
11     }
12     topo.push_back(u);
13     return 1;
14 }
15
16 bool topSort(){
17     if( notInDeg.empty() ) return 0 ;
18     for( int u : notInDeg ){
19         if( !dfs(u) ) return 0;
20     }
```

```

21     reverse( topo.begin() , topo.end() );
22     return 1;
23 }

```

### 3.1.2 Kahn 算法

```

1  vector<int> e[N]; // 邻接表
2  vector<int> topo; // 存储拓扑序
3  int inDeg[N]; // 记录点的当前入度
4
5  bool topSort(){
6      queue<int> q;
7      for( int i = 1 ; i <= n ; i ++ )
8          if( inDeg[i] == 0 ) q.push(i);
9      while( q.size() ){
10         int u = q.front() ; q.pop();
11         topo.push_back(u);
12         for( auto v : e[u] ){
13             if( --inDeg[v] == 0 ) q.push(v);
14         }
15     }
16     return topo.size() == n;
17 }

```

## 3.2 最近公共祖先 (LCA)

### 3.2.1 向上标记法

向上标价法最差的复杂度是  $O(N)$  的，适用于求 LCA 次数少的情况，代码非常好写

```

1  \\ luogu P3379
2  const int N = 5e5+5;
3  int n , m , sta , dep[N] , fa[N];
4  vector<int> e[N];
5
6  int read() {...}
7  void dfs( int x ){
8      for( auto v : e[x] ){
9          if( v == fa[x] ) continue;

```



```

10         dep[v] = dep[x] + 1 , fa[v] =x;
11         dfs(v);
12     }
13 }
14
15 int lca( int x , int y ){
16     if( dep[x] < dep[y] ) swap( x , y );
17     while( dep[x] > dep[y] ) x = fa[x];
18     while( x != y ) x = fa[x] , y = fa[y];
19     return x;
20 }
21
22 int32_t main() {
23     n = read() - 1 , m = read() , sta = read();
24     for( int u , v ; n ; n -- )
25         u = read(),v = read() , e[u].push_back(v) , e[v].push_back(u);
26     dep[sta] = 0 , fa[sta] = sta;
27     dfs( sta );
28     for( int x , y ; m ; m -- ){
29         x = read() , y = read();
30         cout << lca(x,y) << endl;
31     }
32     return 0;
33 }

```

### 3.2.2 倍增 LCA

值得注意的是向上倍增的过程中for( int i = t ; i >= 0 ; i --) 不能写错

```

1  const int N = 5e5+5;
2  int n , m , sta , logN , dep[N] , fa[N][20];
3  vector<int> e[N];
4  int read() {...}
5
6  void dfs( int x ){
7      for( auto v : e[x] ){
8          if( dep[v] ) continue;
9          dep[v] = dep[x] + 1 , fa[v][0] = x;
10         for( int i = 1 ; i <= logN ; i ++ )

```

```

11         fa[v][i] = fa[ fa[v][i-1] ][i-1];
12     dfs(v);
13 }
14 }
15 int lca( int x , int y ){
16     if( dep[x] > dep[y] ) swap( x , y );
17     for( int i = logN ; i >= 0 ; i -- )
18         if( dep[ fa[y][i] ] >= dep[x] ) y = fa[y][i];
19     if( x == y ) return x;
20     for( int i = logN ; i >= 0 ; i -- ){
21         if( fa[x][i] != fa[y][i] ) x = fa[x][i] , y = fa[y][i];
22     }
23     return fa[x][0];
24 }
25
26 int32_t main() {
27     n=read()-1 , m=read() , sta=read() , logN =(int)log2(n) + 1;
28     int k = n;
29     for( int u , v ; n ; n -- )
30         u=read() , v=read() , e[u].push_back(v) , e[v].push_back(u);
31     dep[sta] = 1;
32     dfs( sta );
33     for( int x , y ; m ; m -- ){
34         x = read() , y = read();
35         cout << lca(x,y) << endl;
36     }
37     return 0;
38 }

```

### 3.2.3 Tarjan LCA

### 3.2.4 RMQ 求 LCA

一次 dfs 求出树的深度和欧拉序。区间 $[l,r]$ 中的深度的最小值就是 lca。区间最值用 st 表来维护即可。

## 3.3 最短路

### 3.3.1 Floyd

```

1  int dis[N][N];
2  for( int i = 1 ; i <= n ; i ++ )
3      for( int j = 1 ; j < i ; j ++ )
4          dis[i][j] = dis[j][i] = inf;
5
6  for( int u , v , w ; m ; m -- )
7      u = read() , v = read() , w = read() , dis[u][v] = dis[v][u] = w;
8  for( int k = 1 ; k <= n ; k ++ )
9      for( int i = 1 ; i <= n ; i ++ )
10         for( int j = 1 ; j < i ; j ++ )
11             f[i][j] = f[j][i] = min( f[i][j] , f[i][k] + f[k][j] );

```

### 3.3.2 Dijkstra

复杂度是  $O(m \log(n))$

```

1  void dij(){
2      for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
3      dis[sta] = 0;
4      priority_queue< pair<int,int> , vector<pair<int,int>> , greater<
          pair<int,int>> > q;
5      q.push( { 0 , sta } );
6      while( q.size() ){
7          int u = q.top().second ; q.pop();
8          if( vis[u] ) continue;
9          vis[u] = 1;
10         for( auto [v,w] : e[u] )
11             if( dis[v] > dis[u] + w ){
12                 dis[v] = dis[u] + w;
13                 q.push( {dis[v] , v} );
14             }
15     }
16 }

```

### 3.3.3 Bellman-Ford

复杂度是  $O(km)$

```

1 bool bellmanFord(){ // 返回是否有最短路
2     for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
3     dis[sta] = 0;
4     bool flag;
5     for( int i = 1 ; i <= n ; i ++ ){
6         flag = 0;
7         for( int u = 1 ; u <= n ; u ++ ){
8             if( dis[u] == inf ) continue; // 如果当前点和起点没有联
                通，就无法进行松弛操作
9             for( auto [v,w] : e[u] ){
10                 if( dis[v] <= dis[u] + w ) continue;
11                 dis[v] = dis[u] + w , flag = 1; // 记录时候进行松弛操作
12             }
13         }
14         if( !flag ) break;
15     }
16     return flag;
17 }
```

### 3.3.4 SPFA

没有准确复杂度，下限是  $O(m \log(n))$ ，上限是  $O(nm)$

```

1 int dis[N] , cnt[N];
2 vector< pair<int,int> > e[N];
3 bitset<N> vis;
4
5 bool spfa(){
6     for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
7     queue< int > q;
8     dis[sta] = 0 , vis[sta] = 1 , q.push(sta);
9     for( int u ; q.size() ; ){
10         u = q.front() , q.pop() , vis[u] = 0;
11         for( auto [ v , w ] : e[u] ){
12             if( dis[v] <= dis[u] + w ) continue;
13             dis[v] = dis[u] + w ;
14             cnt[v] = cnt[u] + 1; // 记录最短路经过了几条边
```

```

15         if( cnt[v] >= n ) // 最短路最长是 n-1
16             return false; // 此时说明出现了 负环
17         if( !vis[v] ) vis[v] = 1 , q.push(v);
18     }
19 }
20 return true;
21 }

```

## 3.4 最小生成树

### 3.4.1 Kruskal

Kruskal 总是维护无向图的最小生成森林。

```

1 // Luogu P3366
2 const int N = 5005;
3 int n , m , fa[N] , cnt = 0 , sum;
4
5 int read() {...}
6
7 int getfa( int x ){...}
8
9 void merge( int x , int y ){...} // 并查集合并
10
11 int32_t main(){
12     n = read() , m = read();
13     for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
14     vector<tuple<int,int,int>> e(m);
15     for( auto & [ w , u , v ] : e )
16         u = read() , v = read() , w = read();
17     sort( e.begin() , e.end() );
18     for( auto [ w , u , v ] : e ){
19         if( getfa(u) == getfa(v) ) continue;
20         merge( u , v ) , sum += w , cnt ++;
21         if( cnt == n - 1 ) break;
22     }
23     if( cnt == n - 1 ) cout << sum << "\n";
24     else cout << "orz\n"; // 图不联通
25     return 0;

```

26 }

## 3.5 树

### 3.5.1 树的深度

```
1 dep[sta] = 1;
2
3 void dfs( int u ){
4     for( auto v : e[u] ){
5         if( dep[v] ) continue;
6         dep[v] = dep[u] + 1;
7         dfs( v );
8     }
9 }
10
11 dfs(sta);
```

### 3.5.2 二叉树还原

```
1 struct Node {
2     int v;
3     Node *l, *r;
4     Node(int v, Node *l, Node *r) : v(v), l(l), r(r) {};
5 };
6 // 根据中序、后序还原二叉树
7 Node *build(vector<int> mid, vector<int> suf) {
8     int v = suf.back();
9     Node *l = nullptr, *r = nullptr;
10    int t;
11    for (t = 0; t < mid.size(); t++)
12        if (mid[t] == v) break;
13    auto midll = mid.begin();
14    auto midlr = mid.begin() + t;
15    auto midrl = mid.begin() + t + 1;
16    auto midrr = mid.end();
17    auto sufll = suf.begin();
18    auto suflr = suf.begin() + t;
```

```

19     auto sufrrl = suf.begin() + t;
20     auto sufrr = suf.end() - 1;
21
22     auto midl = vector<int>(midll, midlr);
23     auto midr = vector<int>(midrl, midrr);
24     auto sufl = vector<int>(sufl1, suflr);
25     auto sufr = vector<int>(sufrrl, sufrr);
26
27     if (!midl.empty()) l = build(midl, sufl);
28     if (!midr.empty()) r = build(midr, sufr);
29
30     return new Node(v, l, r);
31 }

```

### 3.5.3 树的 DFS 序和欧拉序

DFS 序中一个点会出现一次，欧拉序会出现两次，两者都是 dfs 时经过的点的顺序，欧拉序中的第二次出现就是点结束搜索是回溯时的顺序

```

1 // 求 DFS 序
2 vector<int> dfsSort;
3 vector<int> e[N];
4 bitset<N> vis;
5
6 void dfs( int x ){
7     dfsSort.push_back(x) , vis[x] = 1;
8     for( auto it : e[x] ){
9         if( vis[it] ) continue;
10        dfs( it );
11    }
12 }
13 // 求欧拉序
14 vector<int> eulerSort;
15 vector<int> e[N];
16 bitset<N> vis;
17
18 void dfs( int x ){
19     eulerSort.push_back(x) , vis[x] = 1;
20     for( auto it : e[x] ){

```

```

21         if( vis[it] ) continue;
22         dfs( it );
23     }
24     eulerSort.push_back(x);
25 }

```

### 3.5.4 树的重心

对于 $size[i]$ 表示每个点子树大小, $max\_part(x)$ 表示删去 $x$ 后最大的子树的大小, $max\_part$ 取到最小值的点 $p$ 就是树的重心

```

1 void dfs( int u ){
2     vis[u] = size[u] = 1;
3     for( auto v : e[u] ){
4         if( vis[v] ) continue;
5         dfs(v);
6         size[x] += size[v];
7         max_part = max( max_part , size[v] );
8     }
9     max_part = max( max_part , n - size[u] );
10    if( max_part > ans )
11        ans = max_part , pos = u;
12 }

```

### 3.5.5 树上前缀和

设 $sum[i]$ 表示节点 $i$ 到根节点的权值总和。

如果是点权,  $x,y$ 路径上的和为 $sum[x]+sum[y]-sum[lca]-sum[fa[lca]]$

```

1 // Loj 2491
2 // 一颗树根节点是 1 , 点权就是深度的 k 次方
3 // m次询问,每次问(u,v)路径上点权之和
4 // k 每次都不同但是取值范围只有[1,50]
5 #define int long long
6 const int N = 3e5+5 , mod = 998244353;
7 int n , sum[N][55] , fa[N][20] , dep[N] , logN;
8 vector<int> e[N];
9
10 int read(){...}
11

```



```

12 void dfs( int x ){
13     for( auto v : e[x] ){
14         if( v == fa[x][0] ) continue;
15         dep[v] = dep[x] + 1 , fa[v][0] = x;
16         for( int i = 1 , val = 1; i <= 50 ; i ++ )
17             val = val * dep[v]% mod , sum[v][i] = (val + sum[x][i]) %
                mod;
18         for( int i = 1 ; i <= logN ; i ++ )
19             fa[v][i] = fa[ fa[v][i-1] ][i-1];
20         dfs(v);
21     }
22 }
23
24 int lca( int x , int y ){...}
25
26 int32_t main(){
27     n = read() , logN = (int)log2(n)+1;
28     for( int i = 2 , u , v ; i <= n ; i ++ )
29         u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u
        );
30     dep[1] = 0;
31     dfs( 1 );
32     for( int m = read() , u , v , k , t ; m ; m -- ){
33         u = read() , v = read() , k = read() , t = lca( u , v );
34         cout << (sum[u][k]+sum[v][k]-sum[t][k]-sum[fa[t][0]][k]+2*mod)
            %mod << "\n";
35     }
36     return 0;
37 }

```

如果是边权,  $x,y$ 路径上的和为 $sum[x]+sum[y]-2*sum[lca]$

```

1 //LOJ 10134 树上前缀和 边
2 const int N = 1e4+5;
3
4 int n , m , sum[N] , logN , dep[N] , fa[N][15];
5 vector<pair<int,int>> e[N];
6
7 int read(){...}
8

```

```

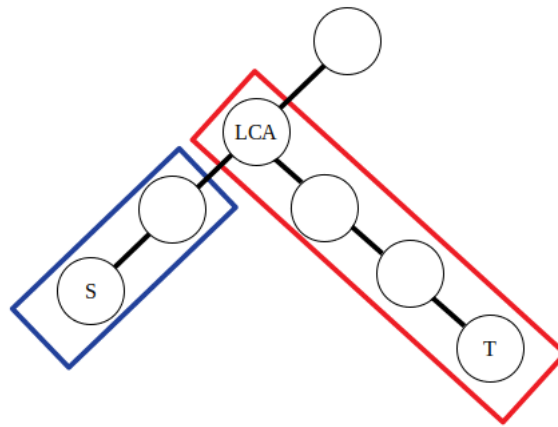
9 void dfs( int x ){ // 这里和 lca 极其类似，就是在多维护了一个前缀和
10     for( auto [v,w] : e[x] ){
11         if( dep[v] ) continue;
12         dep[v] = dep[x] + 1 , fa[v][0] = x , sum[v] = sum[x] + w;
13         for( int i = 1 ; i <= logN ; i ++ )
14             fa[v][i] = fa[ fa[v][i-1] ][ i-1 ];
15         dfs(v);
16     }
17 }
18
19 int lca( int x , int y ){...} // 这里就是 lca 的板子
20
21 int main(){
22     n = read() , m = read() , logN = (int)log2(n)+1;
23     for( int i = 2 , u , v , w ; i <= n ; i ++ )
24         u = read() , v = read() , w = read() , e[u].push_back( {v,w} )
25         , e[v].push_back( {u,w} );
26     dep[1] = 1 , dfs(1);
27     for( int u , v ; m ; m -- ){
28         u = read() , v = read();
29         cout << sum[u] + sum[v] - 2 * sum[ lca(u,v) ] << "\n";
30     }
31     return 0;
32 }

```

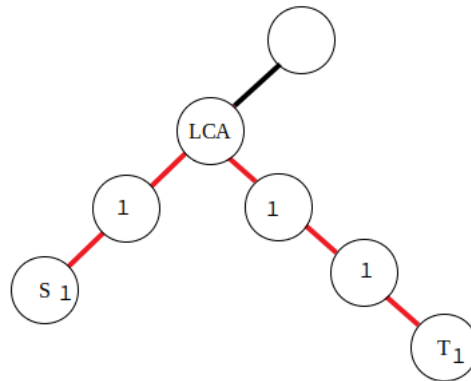
### 3.5.6 树上差分

树上差分就是对树上某一段路径进行差分操作，树上差分分为**点差分**和**边差分**

这里的差分数组用 $d[i]$ 表示，其值是 $a[fa[i]]-a[i]$  点差分



点差分实际上是要对链分成两条链来操作, 如果要对 $(S, T)$ 路径上点权加 $p$  则要 $d[s] += p$  ,  $d[t] += p$  ,  $d[lca] -= p$  ,  $d[fa[lca]] -= p$   
边差分



因为直接对边差分比较困难, 所以要把边权移动到边上的子节点上, 如果要对 $(S, T)$ 路径上边权加 $p$ , 则要 $d[s] += p$  ,  $d[t] += p$  ,  $d[lca] -= 2 * p$

```

1 // Luogu P3128 边差分模板
2 #include<bits/stdc++.h>
3 using namespace std;
4
5 const int N = 5e4+5;
6 int n , m , fa[N][30] , logN , dep[N] , p[N] , res;
7 vector<int> e[N];
8
9 int read(){...}
10

```

```

11 void dfs( int x ) {...} // 和 lca 的 dfs 完全相同
12
13 int lca( int x , int y ){...} // lca
14
15 void getRes( int x ){
16     for( auto v : e[x] ){
17         if( fa[x][0] == v ) continue;
18         getRes(v) , p[x] += p[v];
19     }
20     res = max( res , p[x] );
21 }
22
23 int32_t main(){
24     n = read() , m = read() , logN = log2(n)+1;
25     for( int u , v , i = 1 ; i < n ; i ++ )
26         u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u);
27     dep[1] = 1;
28     dfs( 1 );
29     for( int i = 1 , u , v , d ; i <= m ; i ++ ){
30         u = read() , v = read() , d = lca( u , v );
31         p[u] ++ , p[v] ++ , p[d] -- , p[ fa[d][0] ] --;
32     }
33     getRes( 1 );
34     cout << res << "\n";
35 }

```

## 3.6 图论杂项

### 3.6.1 判断简单无向图图

根据图中的每个点的度可以判断，这个图是不是一个简单无向图（没有重边和自环的无向图）

#### Havel-Hakimi 定理

1. 对当前数列排序，使其呈递减
2. 从  $S[2]$  开始对其后  $S[1]$  个数字-1
3. 一直循环直到当前序列出现负数（即不可简单图化的情况）或者当前序列全为 0（可简单图化）时退出。

```
1 //NC-contest-38105-K
2 int read() {...}
3 priority_queue<int> q; vector<int> ve;
4 int32_t main() {
5     int n = read();
6     for( int i = 1 , x ; i <= n ; i ++ ){
7         x = read() , q.push(x);
8         if( x >= n ) // 如果度数大于 n 一定存在重边或自环
9             cout << "NO\n" , exit(0);
10    }
11    while( 1 ){
12        int k = q.top(); q.pop();
13        if( k == 0 )
14            cout << "YES\n" , exit(0);
15        ve.clear();
16        for( int x ; k ; k -- ){
17            x = q.top() - 1 , q.pop();
18            if( x < 0 )
19                cout << "NO\n" , exit(0);
20            ve.push_back(x);
21        }
22        for( auto it : ve ) q.push(it);
23    }
24    return 0;
25 }
```



## 第四章 数学知识

### 4.1 数论

#### 4.1.1 整除

定义：若整数  $b$  除以非零整数  $a$ ，商为整数且余数为零我们就说  $b$  能被  $a$  整除，或  $a$  整除  $b$  记作  $a|b$

性质

1. 传递性，若  $a|b, b|c$ ，则  $a|c$
2. 组合性，若  $a|b, a|c$  则对于任意整数  $m, n$  均满足  $a|mb + nc$
3. 自反性，对于任意的  $n$ ，均有  $n|n$
4. 对称性，若  $a|b, b|a$  则  $a = b$

#### 4.1.2 约数

定义若整数  $n$  除以整数  $x$  的余数为 0，即  $d$  能整除  $n$ ，则称  $d$  是  $n$  的约数， $n$  是  $d$  的倍数，记为  $d|n$  算数基本定理由算数基本定理得正整数  $N$  可以写作  $N = p_1^{C_1} \times p_2^{C_2} \times p_3^{C_3} \cdots \times p_m^{C_m}$

分解质因数

分解成  $p_1 \times p_2 \times p_3 \times \cdots \times p_n$  这种形式

```
1 vector< int > factorize( int x ){
2     vector<int> ans;
3     for( int i = 2 ; i * i <= x ; i ++){
4         while( x % i == 0 )
5             ans.push_back(i) , x /= i;
6     }
7     if( x > 1 ) ans.push_back(x);
8     return ans;
9 }
```

分解成  $p_1^{k_1} \times p_2^{k_2} \times p_3^{k_3} \times \cdots \times p_n^{k_n}$

```

1 vector< pair<int,int> > factorize( int x ){
2     vector<pair<int,int>> ans;
3     for( int i = 2 , cnt ; i * i <= x ; i ++){
4         if( x % i ) continue;
5         cnt = 0;
6         while( x % i == 0 ) cnt ++ , x /= i;
7         ans.push_back( { i , cnt } );
8     }
9     if( x > 1 ) ans.push_back( { x , 1 } );
10    return ans;
11 }

```

$N$  的正约数个数为 ( $\Pi$  是连乘积的符号, 类似  $\Sigma$ )

$$(c_1 + 1) \times (c_2 + 1) \times \cdots (c_m + 1) = \Pi_{i=1}^m (c_i + 1)$$

$N$  的所有正约数和为

$$(1 + p_1 + p_1^2 + \cdots + p_1^{c_1}) \times \cdots \times (1 + p_m + p_m^2 + \cdots + p_m^{c_m}) = \prod_{i=1}^m \left( \sum_{j=0}^{c_i} (p_i)^j \right)$$

### 4.1.3 GCD 和 LCM

性质  $a \times b = \gcd(a, b) \times (a, b)$

通过性质可以得到最小公倍数的求法就是

```

1 int lcm( int x , int y ){
2     return a / gcd( x , y ) * b;
3 }

```

最大公倍数的求法有更相减损术和辗转相除法

```

1 int gcd( int x , int y ){ // 更相减损术
2     while( x != y ){
3         if( x > y ) x -= y;
4         else y -= x;
5     }
6     return x;
7 }
8
9 int gcd( int x , int y ){ // 辗转相除法
10    return b ? gcd( b , a % b ) : a ;
11 }
12

```



```

13 // 还有一种是直接调用库函数
14 __gcd( a , b );

```

一般情况下直接用库函数，库函数的实现是辗转相除法，如果遇到高精度的话（高精度取模分困难）可以用更相减损术来代替

**定理**对于斐波那契数列  $Feb_i$  有  $Feb_{gcd(a,b)} = gcd(Feb_a, Feb_b)$

#### 4.1.4 质数

判断质数

```

1 bool isPrime( int x ){
2     for( int i = 1 ; i * i <= x ; i ++ )
3         if( x % i == 0 ) return 0;
4     return 1;
5 }

```

埃式筛

```

1 vector< int > prime;
2 bitset<N>notPrime;//不是素数
3
4 void getPrimes( int n ){
5     notPrime[1] = notPrime[0] = 1;
6     for( int i = 2 ; i <= n ; i ++ ){
7         if( notPrime[i] ) continue;
8         prime.push_back(i);
9         for( int j = i * 2 ; j <= n ; j += i )
10             notPrime[j] = 1;
11     }
12 }
13
14 // 如果不需要 prime 数组的话可以优化成下面的代码
15 bitset<N>notPrime;//不是素数
16
17 void getPrimes( int n ){
18     notPrime[1] = notPrime[0] = 1;
19     for( int i = 2 ; i * i <= n ; i ++ ){
20         if( notPrime[i] ) continue;
21         for( int j = i * 2 ; j <= n ; j += i )
22             notPrime[j] = 1;
23     }

```

24 }

欧拉筛

```

1 vector< int > prime;
2 bool notPrime[N];; // 不是素数
3
4 void getPrimes( int n ){
5     notPrime[1] = notPrime[0] = 1;
6     for( int i = 2 ; i <= n ; i ++ ){
7         if( !notPrime[i] ) prime.push_back(i);
8         for( auto it : prime ){
9             if( it * i > n ) break;
10            notPrime[ it * i ] = 1;
11            if( i % it == 0 ) break;
12        }
13    }
14 }
```

**证明质数有无限个**

反证法假设数是  $n$  个，每个素数是  $p_i$ ，令  $P = \prod_{i=1}^n p_i + 1$

因为任何一个数都可以分解成多个质数相乘

所以  $P$  除以任何一个质数都余 1，显然  $P$  就也是一个质数，与假设矛盾，所以假设错误

所以质数是无限个

**性质 2**

设  $\pi(n)$  为不超过  $n$  的质数个数，则  $\pi(n) \approx \frac{n}{\ln n}$

#### 4.1.5 逆元

**费马小定理**

$a^{p-1} \equiv 1(\text{mod } p)$ , 其中  $p$  为素数, 所以  $aa^{p-2} \equiv 1(\text{mod } p)$

```
1 int inv( int x ) {return pow( x , p - 2 );}
```

$O(n)$  递推

```

1 const int N = 1005;
2 int inv[N] = {};
3 void invers(int n,int mod){
4     inv[1] = 1;
5     for(int i = 2;i <= n;i ++) inv[i] = (p-p/i) * inv[p%i] % p;
6     return ;
7 }
```

## 4.2 数学杂项

### 4.2.1 二维向量的叉积

$$\vec{A} \times \vec{B} = |A||B| \cos(\alpha) = a_x \times b_y - a_y \times b_x$$

虽然叉积的结果是一个标量，但是叉积是有正负的，正负取决于向量夹角的大小。

**应用：判断点是否在三角形的内部**对于三角形  $abc$  和一个点  $o$ ， $\vec{ao} \times \vec{ab}$  的正负表示了点  $o$  在线  $ab$  的左侧还是右侧。只要按照顺时针方向（或逆时针方向）判断点  $o$  在三条直线的同一侧，既可以判断点在三角形的内部。

```

1 class Triangle{
2     typedef std::pair<int,int> Vector;
3 private:
4     Vector getVector( int x , int y , int a , int b ){
5         return std::pair{ a - x , b - y };
6     }
7     bool product( Vector a , Vector b ){
8         return ( a.first * b.second - a.second * b.first ) > 0;
9     }
10 public:
11     int ax , ay , bx ,by , cx , cy;
12     Vector ab , bc , ca;
13     Triangle( int ax , int ay , int bx , int by , int cx , int cy ):
14         ax(ax) , ay(ay) , bx(bx) , by(by) , cx(cx) , cy(cy) ,
15         ab( getVector( ax , ay , bx , by ) ),
16         bc( getVector( bx , by , cx , cy ) ),
17         ca( getVector( cx , cy , ax , ay ) ){};
18     Triangle(){};
19     bool isInTriangle( int x , int y ){
20         Vector ao = getVector( ax , ay , x , y );
21         Vector bo = getVector( bx , by , x , y );
22         Vector co = getVector( cx , cy , x , y );
23         bool f1 = product( ao , ab ) , f2 = product( bo , bc ) , f3 =
                product( co , ca );
24         return ( f1 == f2 && f2 == f3 );
25     }
26 };

```

## 4.3 组合数学

### 4.3.1 公式

排列数

$$A_n^n = \frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$$

组合数

$$C_m^n = \frac{m!}{(m-n)! \times n!}$$

组合数性质

1.  $C_n^m = C_n^{n-m}$
2.  $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
3.  $C_n^0 + C_n^1 + \cdots + C_n^n = 2^n$
4.  $C_n^0 + C_n^2 + C_n^4 + \cdots = C_n^1 + C_n^3 + C_n^5 \cdots = 2^{n-1}$
5.  $C_n^m = \frac{n-m+1}{n} \times C_n^{m-1}$

### 4.3.2 组合数计算

```

1 // 暴力计算组合数
2 int C(int x, int y) { // x 中选 y 个
3     y = min(y, x - y);
4     int res = 1;
5     for (int i = x, j = 1; j <= y; i--, j++)
6         res = res * i / j;
7     return res;
8 }
9
10 // 加法递推 O(n^2)
11 for( int i = 0 ; i <= n ; i ++ ){
12     c[i][0] = 1;
13     for( int j = 1 ; j <= i ; j ++ )
14         c[i][j] = c[i-1][j] + c[i-1][j-1];
15 }
16 // 乘法递推 O(n)
17 c[0] = 1;
18 for( int i = 1 ; i * 2 <= n ; i ++ )
19     c[i] = c[n-i] = ( n-i+1 ) * c[i-1] / i;

```

```
20
21 // 任意组合数
22 #define int long long
23 const int N = 5e5+5 , mod = 1e9+7;
24 int fact[N] , invFact[N];
25
26 int power(int x,int y){...} // 快速幂
27 int inv( int x ){...} // 求逆元, 一般是费马小定理
28
29 int A( int x , int y ){ // x 中选 y 排序
30     return fact[x] * invFact[x-y] % mod;
31 }
32
33 int C( int x , int y ){ // x 中选 y 个
34     return fact[x] * invFact[x-y] % mod * invFact[y] % mod;
35 }
36
37 void init(){
38     fact[0] = 1 , invFact[0] = inv(1);
39     for( int i = 1 ; i < N ; i ++ )
40         fact[i] = fact[i-1] * i % mod , invFact[i] = inv(fact[i]);
41 }
```

## 4.4 离散数学



# 第五章 字符串

## 5.1 字符串哈希

### 5.1.1 单哈希

这里用unsigned long long的自然溢出做模数

```
1 typedef unsigned long long ull;
2 const int N = 1e6+5 , K = 1e9+7; // 长度 K进制
3 vector<ull> hashP(N);
4
5 void init(){// 初始化
6     hashP[0] = 1;
7     for( int i = 1 ; i < N ; i ++ ) hashP[i] = hashP[i-1] * K;
8 }
9
10 void hashStr( const string & s , vector<ull> & a){ // 计算Hash数组
11     a.resize(s.size()+1);
12     for( int i = 1 ; i <= s.size() ; i ++ )
13         a[i] = ull(a[i-1] * K + s[i-1]);
14 }
15
16 ull hashStr( const string & s ){
17     ull ans = 0;
18     for( auto i : s )
19         ans = ull( ans * K + i );
20     return ans;
21 }
22
23 ull getHash( int l , int r , const vector<ull> & a ){ //计算Hash值
24     return a[r] - a[l-1] * hashP[r-l+1];
25 }
```

### 5.1.2 双哈希

这里用unsigned long long的自然溢出和998244353做模数

```

1  #define int long long
2  #define mp make_pair
3  typedef pair<int, int> hashv;
4
5  const hashv mod = mp( 1e9+7 , 998244353 );
6  const hashv base = mp(13331,23333);
7
8  hashv operator + ( hashv a , hashv b ) {
9      int c1 = a.first + b.first , c2 = a.second + b.second;
10     if( c1 >= mod.first ) c1 -= mod.first;
11     if( c2 >= mod.second ) c2 -= mod.second;
12     return mp( c1 , c2 );
13 }
14
15 hashv operator - ( hashv a , hashv b ) {
16     int c1 = a.first-b.first , c2 =a.second-b.second;
17     if( c1 < 0 ) c1 += mod.first;
18     if( c2 < 0 ) c2 += mod.second;
19     return mp( c1 , c2 );
20 }
21
22 hashv operator * ( hashv a , hashv b ) {
23     return mp( a.first*b.first%mod.first , a.second*b.second%mod.
        second );
24 }
25
26 const int N = 2e6+5;
27
28 vector< hashv > p , hs , ht;
29
30 void hashStr( const string &s , vector<hashv> &v ){
31     v.resize(s.size()+1);
32     for( int i = 1 ; i <= s.size() ; i ++ )
33         v[i] = v[i-1] * base + mp( s[i-1] , s[i-1] );
34     return;
35 }

```



```
36 hashv getHash( int l , int r , const vector<hashv> &v){
37     if( l > r ) return mp( 0 , 0 );
38     return v[r] - v[l-1] * p[r-l+1];
39 }
40 void init( int n ){
41     p = vector<hashv>( n+1 ) , p[0] = mp(1,1);
42     for( int i = 1 ; i <= n ; i ++ ) p[i] = p[i-1] * base;
43 }
```



# 第六章 动态规划

## 6.1 线性 DP